ΑΡΙΣΤΟΤΕΛΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΟΝΙΚΗΣ
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΠΜΣ ΔΙΚΤΥΑ ΕΠΙΚΟΙΝΩΝΙΩΝ ΚΑΙ ΑΣΦΑΛΕΙΑ ΣΥΣΤΗΜΑΤΩΝ

Κρυπτογραφία

Υλοποιήσεις Συμμετρικών και Ασύμμετρων Αλγορίθμων

Γεώργιος Σ. Ρίζος

ΔΙΔΑΣΚΩΝ: Κωνσταντίνος Δραζιώτης

Αριστοτέλειο Πανεπιστήμιο Θεσσαλονίκης
Σχολή Θετικών Επιστημών
Τμήμα Πληροφορικής

Υπεύθυνη Δήλωση

Βεβαιώνω ότι είμαι συγγραφέας αυτής της εργασίας, και ότι κάθε βοήθεια την οποία είχα για την προετοιμασία της είναι πλήρως αναγνωρισμένη και αναφέρεται στην εργασία. Επίσης έχω αναφέρει τις όποιες πηγές από τις οποίες έκανα χρήση δεδομένων, ιδεών ή λέξεων, είτε αυτές αναφέρονται ακριβώς είτε παραφρασμένες. Επίσης, βεβαιώνω ότι αυτή η εργασία προετοιμάστηκε από εμένα προσωπικά ειδικά για τις απαιτήσεις του μεταπτυχιακού προγράμματος σπουδών Δίκτυα Επικοινωνιών και Ασφάλεια Συστημάτων, του Τμήματος Πληροφορικής του Αριστοτέλειου Πανεπιστημίου Θεσσαλονίκης.

(Υπογραφή)

....

Γεώργιος Σ. Ρίζος

Περιεχόμενα

Μέρος I:

Συμμετρική Κρυπτογραφία

ΚΕΦΑΛΑΙΟ 1

Σύστημα μετατόπισης

Το επόμενο κρυπτόγραμμα έχει ληφθεί:

οκηθμφδζθγοθχυκχσφθμφμχγ

Ο αλγόριθμος κρυπτογράφησης είναι ο εξής: Κάθε γράμμα του αρχικού μας μηνύματος αντικαθίσταται απο την αριθμητική του τιμή ($\alpha \to 1$, ..., $\omega \to 24$). Ας ειναι x_0 μία ρίζα του τριωνύμου $g(x) = x^2 + 3x + 1$. Σε κάθε αριθμό του μηνύματός μου προσθέτω την τιμή του πολυωνύμου $f(x) = x^5 + 3x^4 + 3x^3 + 7x^2 + 3x^4 + 5x + 4$, στο x_0. Αντικαθιστώ κάθε αριθμό με το αντίστοιχο γράμμα. Βρείτε το αρχικό μήνυμα.

1.1 Επίλυση

Αρχικά προσπαθούμε να απλοποιήσουμε το πολυώνυμο $f(x)$ αντικαθιστώντας με το $g(x)$, έτσι ώστε να υπολογίσουμε τις τιμές που μας δίνει το $f(x)$ για x_0. Οπότε με μερικές αντικαταστάσεις καταλήγουμε πως το πολυώνυμο ισούται με:

$$f(x) = (x^3 + 2x + 1) \cdot g(x) + 3$$

Αντικαθιστώντας για $x = x_0$ το $f(x_0) = 3$, καθώς το x_0 είναι ρίζα του $g(x)$, ισχύει $g(x_0) = 0$.

Σύμφωνα με τον αλγόριθμο κρυοπτογράφησης του μηνύματος που παρουσιάζεται στην εκφώνηση, για να παραχθεί το κρυπτογραφημένο μήνυμα έχει προστεθεί στο κάθε γράμμα το $f(x_0)$. Συνεπώς για την αποκρυπτογράφηση του μήνυματος πρέπει να αφαιρεθεί κατά $f(x_0)$, δηλαδή κατά 3 από το κρυπτομήνυμα.

Αφού υλοποιήθηκε ο αντίστοιχος κώδικας σε python, το αποκρυπτογραφημένο μήνυμα είναι το ακόλουθο:

ΜΗΔΕΙΣ ΑΓΕΩΜΕΤΡΗΤΟΣ ΕΙΣΙΤΩ

που αποτελεί τμήμα της φράσης του Πλάτωνα, “Μηδείς αγεωμέτρητος εισίτω μοι την θύρα“.

ΚΕΦΑΛΑΙΟ 2

Το σύστημα του Vigenere

Αποκρυπτογραφήστε το κείμενο [2] εδώ, που κρυπτογραφήθηκε με τον αλγόριθμο του Vigenere.

Υποδ. Για την αποκρυπτογράφηση συστήνουμε να χρησιμοποιήσετε python. Για το μήκος του κλειδιού μπορείτε να χρησιμοποιήσετε είτε test Kasiski ή την μέθοδο του Friedman.

2.1 Επίλυση

Πρωτού προβούμε σε οποιαδήποτε ανάλυση, ελέγχθηκε το κείμενο και διαπιστώθηκε πως υπήρχε ένας αριθμός εντός αυτού τριών ψηφίων, ο αριθμός 100. Ο αριθμός αυτός αφαιρέθηκε καθώς δημιουργούσε προβλήματα κατά την ανάλυση. Το κύριο πρόβλημα προέρχεται από το γεγονός πως ο αριθμός αυτός φαίνεται να μην ειναι κρυπτογραφημένος με το κλειδί. Συνεπώς μετατοπίζει όλα τα δεδομένα μετά απο αυτόν κατα τρείς θέσεις, δυσχεραίνοντας την ανάλυση-ομαδοποίηση που απαιτούταν.

Για τον υπολογισμό του μήκους του κλειδιού χρησιμοποιήθηκαν και οι δύο μέθοδοι που προτείνονται, προς επαλήθευση των αποτελεσμάτων. Συγκεκριμένα από τη μέθοδος του Kasiski προέκυψε ως πιθανότερο κλειδί αυτό με μήκος 7, αφού πρώτα δοκιμάσθηκαν αρκετές λέξεις οι οποίες φάνηκε πως επαναλμβάνονται στο κρυπτογραφημένο μήνυμα. Κατά τη μέθοδο του Friedman τα αποτελέσματα διαφοροποιούνταν ελαφρώς αφού οι τιμές που λαμβάναμε για μήκος κλειδιού ίσο με 7, είχαν αποκλίσεις από τις προτεινόμενες για το αγγλικό αλφάβητο. Πιο αναλυτικά οι τιμές του δείκτη σύμπτωσης κυμαίνονται απο 0.068 εώς 0.072 για μήκος 7, ενώ στα αποτελέσματα των υπολοίπων λήφθηκαν τιμές με μέγιστες τιμές δείκτη κόντα στο 0.05. Θεωρήθηκε λοιπόν πως το μήκος 7 ειναι και αυτό του κλειδιού.

Στη συνέχεια μελετήθηκε η σχετική συχνότητα εμφάνισης του κάθε χαρακτήρα ανά ομάδα, επτά στο σύνολο σύμφωνα με το μήκος κλειδιού. Από κάθε ομάδα λήφθηκε ο χαρακτήρας που εμφανίζεται περισσότερο σ' αυτήν και συγκρίθηκε με τον χαρακτήρα του αγγλικού αλφαβήτου με τον μεγαλύτερο βαθμό εμφάνισης, δηλαδή το ‘E‘. Η διαφορά των δύο χαρακτήρων την ερμηνεύθηκε ως το γράμμα του κλειδιού που γίνεται xor με το αντίστοιχο του μηνύματος και προκύπτει το κρυπτογραφημένο μήνυμα.

Ο πίνακας στο Σχήμα 1 παρουσιάζει τις πιθανές αντιστοιχίες του πιο συχνού γράμματος ανά ομάδα. Στην περίπτωση που το κλειδί κρυπτογράφησης δεν είχε κάποια σημασία και αποτελούταν απλώς από γράμματα χωρίς κάποια έννοια θα ήταν αρκετά δυσκολότερη η εύρεση του σωστού κλειδιού. Με λίγη φαντασία προκύπτει η λέξη κλειδί

‘ SHANNON ‘

είναι αξιοσημείωτο πως η μετατόποιση των γραμμάτων της πέμπτης ομάδος δεν θα είχε βρεθεί χωρίς τη προσθήκη φαντασίας.

Vigenere Candidates Keys		
Group	Most Frequent Cipher Letter	Candidates Letters
1st	L	H, S, V
2nd	P	L, H, W
3rd	T	P, A, I
4th	R	N, X
5th	G	C, J
6th	S	O, D
7th	R	N, C

Σχήμα 1: Υποψήφια κλειδιά του κρυπτοσυστήματος Vigenere.

ΚΕΦΑΛΑΙΟ 3

Κυκλική Κύλιση

Έστω ένα μήνυμα m, 16-bits. Θεωρούμε την κυκλική κύλιση προς τα αριστερά << a κατά a bits. Έστω,

$$c = m \oplus (m \lll 6) \oplus (m \lll 10)$$

Βρείτε τον τύπο αποκρυπτογράφησης. Υλοποιήστε κατάλληλο κώδικα για να δείξετε ότι ο τύπος που φτιάξατε είναι σωστός.

3.1 Επίλυση

Αρχικά μελετήθηκε η μετακίνηση των bits του κάθε μηνύματος βάση του αρχικού. Αναφερόμαστε σε μηνύματα, προς ακρίβεια τρία, καθώς θεωρήθηκε πως για την παραγωγή του κρυπτογραφημένου μηνύματος γίνονται xor τρία μηνύματα, το αρχικό, το αρχικό με κυκλική κύλιση κατά 6-bits και κατά 10-bits. Τα μηνύματα παρουσιάζονται στο Σχήμα 2 διαχωρισμένα, χρωματισμένα κατά ομάδες bits σύμφωνα με τη κύλιση που λαμβάνει χώρα και αριθμημένα βάση της θέσης του κάθε bit στην ομάδα που ανήκει. Έπειτα τα μηνύματα διαχωρίσθηκαν σε 4 τετράδες, καθώς το μικρότερο ακέραιο τμήμα μηνύματος μετά την κάθε κύλιση είναι 4-bits, Σχήμα 3, αυτές οι τετράδες ονοματίσθηκαν για λόγους ευκολίας, Σχήμα 4.

m<<10	1234561234561234
m<<6	1234123456123456
m	1234561234123456

Σχήμα 2: Μηνύματα τμηματοποιημένα.

Προκειμένου να καταλήξουμε στον τρόπο που παράγεται το κρυπτογραφημένο μήνυμα, αναζητήθηκαν κάποιες σχέσεις οι οποίες οδηγούν σ' αυτό. Οι σχέσεις που παρουσιάζονται στη συνέχεια ισσούνται με τις επιμέρους τετράδες bits του κρυπτογραφημένου μηνύματος.

m<<10	1234 5612 3456 1234
m<<6	1234 1234 5612 3456
m	1234 5612 3412 3456

m<<10	12 3456 1234 5612 34
m<<6	12 3412 3456 1234 56
m	12 3456 1234 1234 56

Σχήμα 3: Μηνύματα διαχωρισμένα κατά ομάδες 4-bits.

x	1234
y	1234
z	1234
w	5612
l	3456
k	3456
m	5612
5	3412

Σχήμα 4: Ονοματολογία ομάδων.

1. $x \oplus y \oplus z$
2. $w \oplus x \oplus m$
3. $l \oplus w \oplus t$
4. $y \oplus l \oplus k$
5. $k \oplus t \oplus l$
6. $z \oplus k \oplus y$
7. $m \oplus z \oplus x$

Με πράξεις μεταξύ αυτών προκύπτουν και οι παρακάτω σχέσεις:

- $m \oplus y$
- $w \oplus z$
- $x \oplus k$
- $w \oplus k$
- $z \oplus l$
- $y \oplus t$
- $z \oplus k$
- $m \oplus t$
- $w \oplus l$

Η αρχή του μίτου της αποκάλυψης του μηνύματος, γίνεται με την εύρεση του y, το οποίο ισούται με $z \oplus k \oplus z \oplus k \oplus y$. Αναλυτικότερα προκύπτουν ακολούθως:

- $w \oplus z = w \oplus x \oplus m \oplus m \oplus z \oplus x$ (2), (7)
- $w \oplus k = l \oplus w \oplus t \oplus k \oplus t \oplus l$ (3), (5)
- $z \oplus k = w \oplus z \oplus w \oplus k$
- $y = z \oplus k \oplus z \oplus k \oplus y$ (6)

Αντίστοιχα υπολογίζεται το m και το t:

- $m \oplus y = x \oplus y \oplus z \oplus m \oplus z \oplus x$ (1), (7)

- $m = y \oplus m \oplus y$
- $t \oplus y = y \oplus l \oplus k \oplus k \oplus t \oplus l$ (4), (5)
- $t = y \oplus t \oplus y$

m<<10	12 34561234 561234
m<<6	123412 3456123456
m	1234 56123412 3456

Σχήμα 5: Γραμμοσκιασμένα γνωστά bits του μηνύματος.

Μέχρι στιγμής γνωρίζουμαι 8-bits του αρχικού μηνύματος, όπως φαίνεται και στο Σχήμα 5. Για την αποκάλυψη και των υπολοίπων θα χρειαστεί να χωρίσουμε τα μηνύματα, όπως έχουν αναφερθεί, σε ομάδες των 2-bits τις οποίες θα κάνουμε xor μεταξύ τους, εκμεταλευόντας όπως και πριν μερικές από τις ιδιότητες της πράξης. Έστω:

- $y1 = y_1||y_2$
- $y2 = y_3||y_4$
- $m1 = m_1||m_2$
- $t2 = t_3||t_4$
- $c_i = cipher_{2i}||cipher_{2i+1}$, $i\epsilon[1, 16]$, όπου $cipher_1$ το πρώτο-bit του κρυπτογραφημένου μηνύματος
- $m_i = message_{2i}||message_{2i+1}$, $i\epsilon[1, 16]$, όπου $message_1$ το πρώτο-bit του μηνύματος

Με τις ακόλουθες πράξεις καταλήγουμε στο τελικό μήνυμα.

- $m_1 = t2 \oplus y1 \oplus c_1$
- $m_7 = m_1 \oplus y1 \oplus c_4$
- $m_2 = m_7 \oplus y1 \oplus c_7$
- $m_8 = t2 \oplus m1 \oplus c_3$

Η κρυπτογράφιση και αποκρυπτογράφιση του μηνύματος ακολουθώντας την παραπάνω μεθοδολογία, αποδεικνύεται με την υλοποιήση του ανάλογου κώδικα σε python.

ΚΕΦΑΛΑΙΟ 4

One Time Pad

Υλοποιήστε τον OTP αφού αρχικά μετατρέψετε το μήνυμα σας σε bit με χρήση του παρακάτω πίνακα. Θα πρέπει να δουλεύει η κρυπτογράφηση και η αποκρυπτογράφηση. Το μήνυμα δίνεται κανονικά και έσωτερικά μετατρέπεται σε bits. Το κλειδί είναι διαλεγμένο τυχαί και έχει μήκος όσο το μήκος του μηνύματος σας. Το αποτέλεσμα δίνεται όχι σε bits αλλά σε λατινικούς χαρακτήρες.

4.1 Επίλυση

Στο πρόγραμμα που αναπτύχθηκε σε python, δημιουργήθηκε μία μέθοδος με όνομα "alphabet" που αναλαμβάνει την μετατροπή οποιουδήποτε χαρακτήρα στον αντίστοιχο δυαδικό αριθμό που προκύπτει βάση του δοθέντος πίνακα. Η μέθοδος "numberbet" υλοποιοεί το αντίστροφο για την εκτύπωση των αποτελέσματων στη συνέχεια. Η μέθοδος xor είναι αυτή που αναλαμβάνει την υλοποίηση της πράξης xor μεταξύ δύο δυαδικών αριθμών. Επίσης η μέθοδος "to_digit" χρησιμοποιώντας τα παραπάνω μετατρέπει το κείμενο που δέχεται ως είσοδο σε δυαδικό.

Η παραγωγή του κλειδιού γίνεται με τη μέθοδο key η οποία δέχεται ως είσοδο των αριθμό των bits που θα θα είναι το κλειδί. Έπειτα υπολογίζει το μέγιστο αριθμό που μπορεί να πάρει το κλειδί με τα δοθέν αριθμό bits και με χρήση της βιβλιοθήκης random της python παράγει τον ψευδοτυχαίο αριθμό που θα οριστεί ως κλειδί. Τέλος αυτός θα μετατραπεί σε bits, θα γίνει xor με το μήνυμα εισόδου και θα παραχθεί το κρυπτογραφημένο μήνυμα, η αντίστροφη διαδικασία είναι η διαδικασία αποκρυπτογράφησης.

ΚΕΦΑΛΑΙΟ 5

RC4

Υλοποιήστε τον RC4. Χρησιμοποιώντας το κλειδί HOUSE κρυπτογραφήστε το μήνυμα (ξαναγράψτε το χωρίς κενά):

MISTAKES ARE AS SERIOUS AS THE RESULTS THEY CAUSE

Η υλοποίηση σας πρέπει και να αποκρυπτογραφεί σωστά. Χρησιμοποιήστε τον πίνακα της άσκησης 3.5.

5.1 Επίλυση

Η υλοποίηση του RC4 έγινε σε python και χρησιμοποιεί τις βασικές μεθόδους μετατροπών αλφαριθμητικών χαρακτήρων σε bytes. Η τροποποιήση που δέχθηκαν οι μέθοδοι αυτοί αφορούν τα bytes καθώς η άσκηση 3.5 έκανε τη μετατροπή σε bits και αντιστρόφως. Οι κύριες μέθοδοι που αναπτύχθηκαν είναι αυτή της μετάθεσης "shift" και της κλειδορoής RC4. Η μέθοδος της μετάθεσης δέχεται ως είσοδο το κλειδί και επιστρέφει την αρχική μετάθεση δηλαδή έναν πίνακα S 256-bits, εν προκειμένω 32 θέσεων αν υποθέσουμε πως κάθε θέση ειναι 1-byte. Η μέθοδος της κλειδορoής εξάγει το κλειδί βάση του πίνακα μετάθεσης που δέχεται ως είσοδο. Το κλειδί αυτό γίνεται xor με το μήνυμα ενώ για την αποκρυπτογράφηση του μηνύματος ακολουθείται η αντίστροφη διαδικασία.

Το κρυπτογραφημένο μηνυμα που παράγεται είναι το ακόλουθο:

M.CISGLBIIJSFQ-ANTP.A-DCVIVTPZOB((JDO!XLY

φυσικά τα κενά παραλείπονται.

ΚΕΦΑΛΑΙΟ 6

Differential Uniformity

Αν Σ ένα σύνολο με |Σ| συμβολίζουμε το πλήθος των στοιχείων του. Υπολογίστε τη διαφορική ομοιομορφία (differential uniformity) του S-box (6),

$$Diff(S) = max_{x\epsilon\{0,1\}^6-\{0\},y\epsilon\{0,1\}^4} |z \epsilon \{0,1\}^6 : S(z \oplus x) \oplus S(z) = y|$$

Γενικά, για S-boxes:

$$S : \{0,1\}^n \to \{0,1\}^m$$

και ισχύει:

$$Diff(S) \geq max\{2, 2^{n-m}\}$$

6.1 Επίλυση

Για το S-box (6) ισχύει:

$$S : \{0,1\}^6 \to \{0,1\}^4$$

και επίσης σύμφωνα με την τελευταία ανισότητα προκύπτει:

$$Diff(S) \geq max\{2, 2^{n-m}\} = max\{2, 2^{6-4}\} = max\{2, 4\} = 4 \Rightarrow Diff(S) \geq 4$$

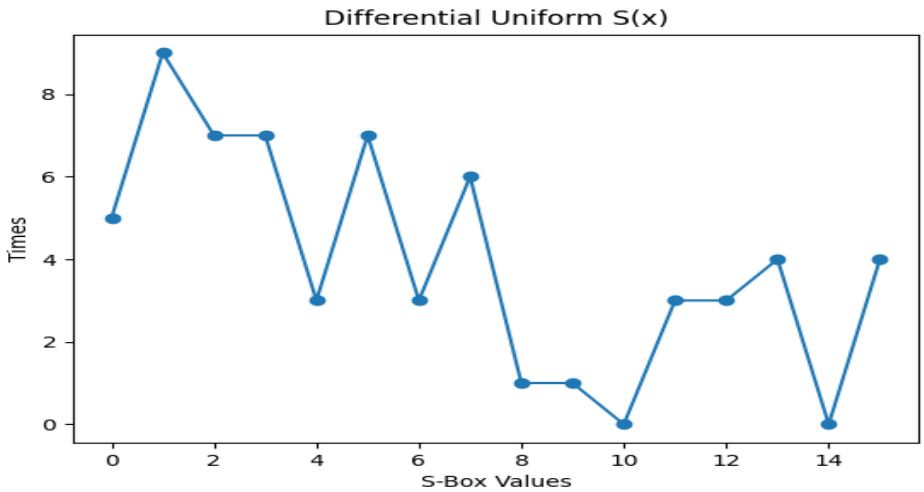
Για τον υπολογισμό του βαθμού της διαφορικής ομοιομορφίας του S-box που μελετήθηκε, αναπτύχθηκε αλγόριθμος σε python ο οποίος υπολογίζει για όλες τις πιθανές τιμές του x, τα αποτελέσματα που δίνει το S-box. Το διάγραμμα των αποτελεσμάτων παρουσιάζεται στο Σχήμα 7, και όπως φαίνεται η μέγιστη τιμή που παίρνουμε είναι το 9, ενώ τα πιθανά x είναι $|x| = 64 - \{0\} = 63$. Βάση του διαγράμματος μπορεί να υποθεί πως τα αποτελέσματα δεν ακολουθούν κάποια ομοιόμορφη κατανομή.

Στη συνέχεια υπολογίσθηκαν οι τιμές που μπορεί να πάρει το y, όπου

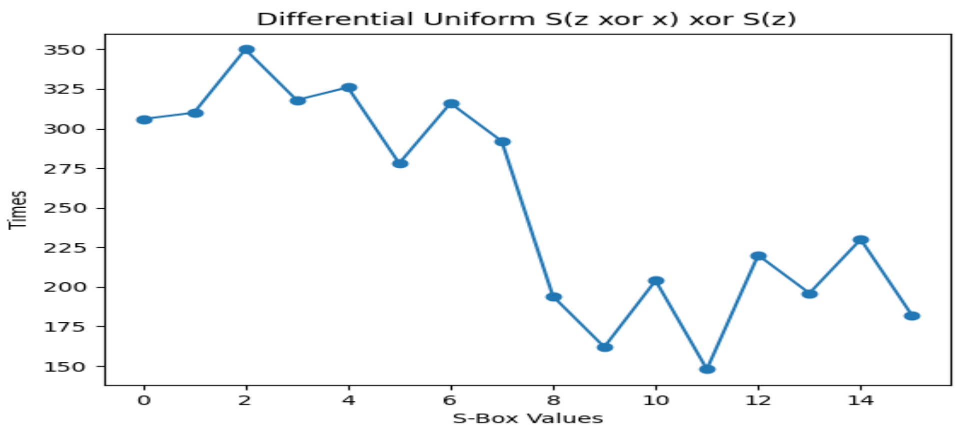
$$y = S(z \oplus x) \oplus S(z)$$

και η απεικόνιση του αντίστοιχου διαγράμματος παρουσιάζεται στο Σχήμα 8. Εδώ παρατηρούμε πως η κατανομή των αποτελεσμάτων είναι αντίστοιχη του προηγούμενο σχήματος, με μέγιστο πλήθος

$$max_{x\epsilon\{0,1\}^6-\{0\},y\epsilon\{0,1\}^4} |y| = 350$$



Σχήμα 6: Differential Uniform για $S(x)$.



Σχήμα 7: Differential Uniform για $S(z \oplus x) \oplus S(z) = y$.

Για την υλοποίηση της άσκησης χρησιμοποιήθηκαν δύο βασικές βιβλιοθήκες της python, η numpy και η matplotlib. Η εγκατάσταση και των δύο γίνεται πολύ εύκολα με χρήση του pip installer, ένα εργαλείο που διευκολύνει την εγκατάσταση των περισσότερων πακέτων της python. Η βιβλιοθήκη numpy αποτελεί ένα πολύ χρήσιμο εργαλείο, που σου δυνατότητες επεξεργασίας πινάκων με ποικίλους τρόπους, ενώ η matplotlib είναι βιβλιοθήκη για αναπαράσταση διαφόρων γραφημάτων.

ΚΕΦΑΛΑΙΟ 7

Avalanche Effect

Εξετάστε αν ισχύει το avalanche effect στο AES. Αναλυτικότερα, φτιάξτε αρκετά ζευγάρια (> 30) μηνυμάτων (m_1, m_2) που να διαφέρουν σ ένα bit. Εξετάστε σε πόσα bits διαφέρουν τα αντίστοιχα κρυπτομηνύματα. Δοκιμάστε με δύο καταστάσεις λειτουργίας: ECB και CBC. Τα μήκη των μηνυμάτων που θα χρησιμοποιήσετε να έχουν μήκος διπλάσιο του μήκους ενός block. Δηλ. για τον AES, να είναι μήκους 256-bits.

7.1 Επίλυση

Ο υπολογισμός του avalanche effect και για τις δύο καταστάσεις λειτουργίας έγινε με τη δημιουργία και σύγκριση 100 ζευγαριών για κάθε μία κατάσταση. Το κάθε ζευγάρι μηνυμάτων κρυπτογραφόταν με το ίδιο κλειδί το οποίο διέφερε απο ζευγάρι σε ζευγάρι. Το κλειδί παραγόταν μέσω της μεθόδου key_generator δίνοντας ως είσοδο το μέγεθος του κλειδιού, δηλαδή 128-bits ίσο με το μήκος του κάθε μηνύματος, μέσω της random μια βιβλιοθήκη της python. Ανάλογη διαδικασία ακολουθήθηκε και για την παραγωγή κάθε μηνύματος, ενώ κάθε δεύτερο μήνυμα διαφοροποιόταν σ' ένα ψευδό-τυχαίο bit απ' το πρώτο. Στο τέλος τα μηνύματα μετατρέποταν σε bytes ώστε να κρυπτογραφηθούν με τον AES, το κρυπτογραφημένο μήνυμα μετατρεπόταν ξανά σε bits ώστε να γίνει η σύκγριση.

Οι μέσοι όροι των αποτελεσμάτων είναι οι ακόλουθοι σύμφωνα με τις καταστάσεις λειτουργίας:

CBC: 128-bits
ECB: 64-bits

Όπως εξάγεται εκ των αποτελεσμάτων, στην κατάσταση CBC ισχύει το avalanche effect καθώς η αλλαγή ενός bit εισόδου φέρει αλλαγή στο 50% του μηνύματος, άρα κάθε bit αλλάζει με πιθανότητα 50%. Εν αντιθέση στην κατάσταση CBC προκύπτει αλλαγή στο 25% του μηνύματος, συνεπώς δεν ισχύει το avalanche effect.

Η υλοποίηση της άσκησης εκμεταλεύτηκε στη βιβλιοθήκη της python, Cryptodome η οποία περιέχει ποικίλους αλγόριθμος κρυπτογράφησης, και εφόσον το ζητούμενο της άσκησης ήταν η αξιολόγηση του AES ως προς το avalanche effect χρησιμοποιήθηκε μία έτοιμη υλοποίηση του αλγορίθμου. Επίσης χρησιμοποιήθηκε και η βιβλιοθήκη numpy που αναφέρθηκε σε προηγούμενο κεφάλαιο. Η εγκατάσταση και των δύο μπορεί να γίνει πολύ εύκολα μέσω του pip installer.

Μέρος II:

Ασύμμετρη Κρυπτογραφία

ΚΕΦΑΛΑΙΟ 8

Diffie Helman

Σας δίνεται ότι το $g^{ab} \mod p = 1$ και $g = 13, p = 677$. Υπολογίστε τα υποψήφια a, b. Θα χρειαστεί να υλοποιήσετε τον αλγόριθμο της γρήγορης ύψωσης σε δύναμη (2.2.3).

8.1 Επίλυση

Υλοποιήθηκε ο ζητούμενος αλγόριθμος και προς επαλήθευση της ταχύτητάς του υλοποιήθηκε και ο παραδοσιακός αλγόριθμος ύψωσης σε δύναμη. Σύμφωνα με τον τελευταίο πολλαπλασιάζουμε τη βάση με τον εαυτό της τόσες φορές όσο και το μέγεθος του εκθέτη. Στην εικόνα 1 παρουσιάζεται ο χρόνος εκτέλεσης της πράξης $2^{1234567} mod 157$ με χρήση και των δύο αλγορίθμων, είναι φανερό πως ο αλγόριθμος (2.2.3) υπερέχει του παραδοσιακού.

Προς επίλυση του ζητούμενο υπολογίσθηκαν τα πιθανά a, b τα οποία επαληθεύουν τη σχέση $g^{ab} \mod p = 1$, με εφαρμογή αλγορίθμου ωμής βίας. Το πλήθος των πιθανών a, b είναι το 7688, σ' αυτό δε συμπεριλήφθηκε η μονάδα ενώ θεωρήθηκαν δεκτά τα αποτελέσματα όταν $a = b$. Ο χρόνος εκτέλεσης είναι περίπου 4 sec, όπως φαίνεται και στην εικόνα 2 .

```
Starting
         3 function calls in 0.000 seconds

   Random listing order was used

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        1    0.000    0.000    0.000    0.000 /Users/rizos/Desktop/crypto2/1-to_pow.py:28(to_pow)
        1    0.000    0.000    0.000    0.000 /Users/rizos/Desktop/crypto2/1-to_pow.py:45(<lambda>)



         3 function calls in 31.919 seconds

   Random listing order was used

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        1   31.919   31.919   31.919   31.919 /Users/rizos/Desktop/crypto2/1-to_pow.py:19(traditional_to_pow)
        1    0.000    0.000   31.919   31.919 /Users/rizos/Desktop/crypto2/1-to_pow.py:46(<lambda>)



New Method: 2^1234567 mod 157 = 116
Traditional: 2^1234567 mod 157 = 116
```

Σχήμα 8: Χρόνος εκτέλεσης αλγορίθμων ύψωσης σε δύναμη.

```
         918943 function calls in 4.031 seconds

   Random listing order was used

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     7688    0.001    0.000    0.001    0.000 {method 'append' of 'list' objects}
        1    0.001    0.001    0.001    0.001 {built-in method fromkeys}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        1    0.000    0.000    0.000    0.000 /Users/rizos/Desktop/crypto2/1/1-to_pow.py:20(<listcomp>)
        1    0.142    0.142    4.031    4.031 /Users/rizos/Desktop/crypto2/1/1-to_pow.py:15(keys_finder)
        1    0.000    0.000    4.031    4.031 /Users/rizos/Desktop/crypto2/1/1-to_pow.py:45(<lambda>)
   455625    0.124    0.000    3.888    0.000 ../utilities.py:1(pow)
   455625    3.763    0.000    3.763    0.000 ../utilities.py:34(pow_modulus)
```

Σχήμα 9: Χρόνος εκτέλεσης υποψήφιων a, b σε second.

ΚΕΦΑΛΑΙΟ 9

Karatsuba

Υλοποιήστε σε όποια γλώσσα προγραμματισμού θέλετε τον αλγόριθμο 2.2.2 και κατόπιν υπολογίστε το γινόμενο
$$2^{1000} \cdot 3^{101} \cdot 5^{47} (\mod 2^{107} - 1)$$
κάνοντας χρήση του αλγόριθμου του Karatsuba.

9.1 Επίλυση

Ο αλγόριθμος του Karatsuba υλοποιήθηκε σε Python και προς επαλήθευση της ταχύτητας του αναπτύχθηκε και ο παραδοσιακός αλγόριθμος πολλαπλασιασμού. Ο αλγόριθμος του Karatsuba μείωσε το κόστος του πολλαπλασιασμού σε $O(n^{1.5})$, πιο συγκεκριμένα για το το πολλάπλασιασμό δύο αριθμών τεσσάρων (4) bytes απαιτεί έναν πολλαπλασιασμό και τρείς προσθέσεις σε σύγκριση με τη παραδοσιακή μέθοδο που απαιτεί τέσσερις πολλαπλασιασμούς και μία πρόσθεση. Στην εικόνα 3, συγκρίνεται ο χρόνος εκτέλεσης των δύο αλγορίθμων υπολογίζοντας το γινόμενο
$$3^{12345} \cdot 5^{12345}$$
ενώ η πράξη της ύψωσης σε δύναμη γίνεται με τον αλγόριθμο της προηγούμενης ενότητας.

Το αποτέλεσμα της πράξης που ζητήθηκε στην εκφώνηση
$$2^{1000} \cdot 3^{101} \cdot 5^{47} (mod 2^{107} - 1) = 32650406043484327258729325887 91$$
όπως και ο χρόνος εκτέλεσης σε seconds παρουσιάζονται στην εικόνα 4.

```
        11608744 function calls (8183041 primitive calls) in 7.003 seconds

   Random listing order was used

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
  7041137    0.487    0.000    0.487    0.000 {built-in method builtins.len}
  1141901    0.259    0.000    0.259    0.000 {built-in method builtins.max}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        1    0.000    0.000    7.003    7.003 /Users/rizos/Desktop/crypto2/2/2-exercise.py:34(<lambda>)
3425704/1    6.256    0.000    7.003    7.003 ../utilities.py:98(karatsuba)



        7 function calls in 1409.623 seconds

   Random listing order was used

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        2    0.000    0.000    0.000    0.000 {method 'reverse' of 'list' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        1    0.001    0.001    0.001    0.001 /Users/rizos/Desktop/crypto2/2/2-exercise.py:6(<listcomp>)
        1    0.002    0.002    0.002    0.002 /Users/rizos/Desktop/crypto2/2/2-exercise.py:7(<listcomp>)
        1 1409.620 1409.620 1409.623 1409.623 /Users/rizos/Desktop/crypto2/2/2-exercise.py:5(traditional_multiplication)
        1    0.000    0.000 1409.623 1409.623 /Users/rizos/Desktop/crypto2/2/2-exercise.py:35(<lambda>)
```

Σχήμα 10: Συγκρίση παραδοσιακής μεθόδου πολλαπλασιασμού με μέθοδο του Karatsuba.

```
         44711 function calls (31592 primitive calls) in 0.027 seconds

   Random listing order was used

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    27206    0.002    0.000    0.002    0.000 {built-in method builtins.len}
     4373    0.001    0.000    0.001    0.000 {built-in method builtins.max}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        1    0.000    0.000    0.027    0.027 /Users/rizos/Desktop/crypto2/2/2-exercise.py:20(solve)
        1    0.000    0.000    0.027    0.027 /Users/rizos/Desktop/crypto2/2/2-exercise.py:41(<lambda>)
        4    0.000    0.000    0.000    0.000 ../utilities.py:1(pow)
        4    0.000    0.000    0.000    0.000 ../utilities.py:14(pow_regular)
  13121/2    0.024    0.000    0.027    0.014 ../utilities.py:139(karatsuba)



2^100 * 3^101 * 5^47 (mod 2^107 - 1) = 32650406043484327258729325887 91
```

Σχήμα 11: Υπολογισμός του $2^{1000} \cdot 3^{101} \cdot 5^{47} (mod 2^{107} - 1)$.

ΚΕΦΑΛΑΙΟ 10

Αριθμός του Πλάτωνα

Έστω m θετικός ακέραιος και $\sigma(m)$ το άθροισμα των θετικών διαιρετών του m. Να αποδείξετε ότι ο μεγαλύτερος ακέραιος n μέσα στο διάστημα $[2, 10^7]$ με την ιδιότητα

$$\sigma(n) > e^{\gamma} n \ln(\ln n)$$

είναι ο 5040. Η σταθερά γ του Euler είναι $\gamma \simeq 0.577\ldots$

Σχόλια. Ο αριθμός 5040 ονομάζεται και αριθμός του Πλάτωνα και εμφανίζεται πρώτη φορά στο 5ο βιβλίο *των Νόμων* του Πλάτωνα, ως ο ιδανικός αριθμός οικογενειών μίας ιδανικής πόλης.

10.1 Επίλυση

Αναπτύχθηκε αλγόριθμος σε python ο οποίος αφού υπολογίσει το άθροισμα όλων των διαιρετών ενός αριθμού n όπου $n \in [2, 10^7]$ τον συγκρίνει με τη δοθείσα συνθήκη. Το άθροισμα των διαιρετών του 5040 είναι $\sigma(5040) = 19344$, και το πλήθος τους 60 συμπεριλαμβανομένου και του ιδίου. Αποδυκνύεται ότι ισχύει $\sigma(5040) > e^{\gamma} \cdot 5040 \cdot \ln(\ln 5040)$. Μάλιστα

$$e^{\gamma} \cdot 5040 \cdot \ln(\ln 5040) = 5101.90\ldots$$

Παρόλα αυτά δεν αποδείχθηκε η μοναδικότητα του. Σύμφωνα με τον αλγόριθμο εμφανίζονται και άλλοι αριθμοί οι οποίοι πληρούν τη συνθήκη, όπως φαίνεται και στην εικόνα 5. Φυσικά ο αριθμός 5040 χαρακτηρίζεται από το μεγάλο πλήθος των διαιρετών του αλλά και του αθροίσματος τους, όμως στο σύνολο αναζήτησης ο αμέσως επόμενος αριθμός με μεγαλύτερο άθροισμα διαιρετών ειναι ο 5760 με $\sigma(5760) = 19890 > \sigma(5040)$, αντίστοιχα συμβαίνει και με τη συνθήκη.

```
5030 -> sum: 9072 _ condition: 5090.952761134636
5031 -> sum: 8008 _ condition: 5092.0556210972545
5032 -> sum: 10260 _ condition: 5093.158496978566
5033 -> sum: 5760 _ condition: 5094.26138877509
5034 -> sum: 10080 _ condition: 5095.364296483339
5035 -> sum: 6480 _ condition: 5096.467220099833
5036 -> sum: 8820 _ condition: 5097.570159621089
5037 -> sum: 7104 _ condition: 5098.673115043629
5038 -> sum: 8280 _ condition: 5099.776086363974
5040 -> sum: 19344 _ condition: 5101.9820766841785
5041 -> sum: 5113 _ condition: 5103.085095677087
5042 -> sum: 7566 _ condition: 5104.188130553905
5043 -> sum: 6892 _ condition: 5105.291181311159
5044 -> sum: 9604 _ condition: 5106.3942479453835
5045 -> sum: 6060 _ condition: 5107.497330453107
5046 -> sum: 10452 _ condition: 5108.600428830865
5047 -> sum: 5928 _ condition: 5109.703543075191
5048 -> sum: 9480 _ condition: 5110.806673182627
5049 -> sum: 8640 _ condition: 5111.909819149703
5050 -> sum: 9486 _ condition: 5113.012980972962
5052 -> sum: 11816 _ condition: 5115.219352174199
5053 -> sum: 5248 _ condition: 5116.322561545261
5054 -> sum: 9144 _ condition: 5117.4257867586775
5055 -> sum: 8112 _ condition: 5118.529027810996
5056 -> sum: 10160 _ condition: 5119.632284698767
5057 -> sum: 5460 _ condition: 5120.735557418536
5058 -> sum: 10998 _ condition: 5121.838845966856
```

Σχήμα 12: Αποτελέσματα της εκτέλεσης $\sigma(n) > e^{\gamma} n \ln(\ln n)$.

ΚΕΦΑΛΑΙΟ 11

Τεστ Πιστοποίησης Πρώτων

Να εξετάσετε αν ο αριθμός Fibonacci F_{104911} είναι ισχυρός πιθανός πρώτος (δείτε άσκηση 3.33).
Υποδ. Ο αριθμός αυτός έχει 21925 ψηφία. Θα χρειαστείτε έναν κατάλληλο αλγόριθμο που να μπορεί να υπολογίζει δυνάμεις $\mod p$ για πολύ μεγάλους ακεραίους.

11.1 Επίλυση

Αναπτύθηκε σειριακός αλγόριθμος για τον υπολογισμό του αριθμού F_{104911} ο οποίος έχει πολυπλοκότητα $O(n)$ και απαιτήσεις μνήμης $O(1)$. Επιλέχθηκε ο συγκεκριμένος αλγόριθμος [2] καθώς ο αριθμός της ακολουθίας που αναζητάμε είναι 21925 ψηφίων, έτσι με το σύνηθες αναδρομικό αλγόριθμο ενδεχομένως να αντιμετοπίζει προβλήματα μνήμης ένα σύστημα με πενιχρούς πόρους μνήμης.

Προς απόδειξη της διαιρετότητας του αριθμού αυτού χρησιμοποιήθηκε το τεστ πιστοποιήσης πρώτων των Miller Rabin. Ενώ για τον γρήγορο υπολογισμό δυνάμεων, χρησιμοποιήθηκε ο αλγόριθμος γρήγορης ύψωσης σε δύναμη που υλοποιήθηκε στο 1ο Κεφάλαιο, αν και παρατηρήθκε πως παρέμεινε η πιο υπολογιστικά κοστοβόρα διαδικασία. Ο αριθμός αποδείχθηκε πως είναι ισχυρός πιθανός πρώτος μετά από αρκετό χρόνο σε περίπου μία ώρα, καθώς επαληθεύτηκε τέσσερις (4) φορές. Στην εικόνα 6 παρουσιάζονται τα αποτελέσματα.

```
/usr/local/bin/python3.6 /Users/rizos/Desktop/crypto2/miller_rabin.py
Prime Search for Fibonacci(104911)
Number of evaluation 4-times.

START
s: 2
t-size: 21925 digits

* * *         56 function calls in 4077.982 seconds

   Random listing order was used

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        4    0.000    0.000    0.000    0.000 {method 'bit_length' of 'int' objects}
        1    0.000    0.000    0.000    0.000 {built-in method builtins.hasattr}
        5    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        1    0.000    0.000    0.000    0.000 {built-in method math.log10}
        1    0.000    0.000    0.000    0.000 {built-in method math.log2}
        7    0.000    0.000    0.000    0.000 {method 'getrandbits' of '_random.Random' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        1    0.000    0.000    0.000    0.000 <frozen importlib._bootstrap>:997(_handle_fromlist)
        4    0.000    0.000    0.000    0.000 /Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/random.py:173(randrange)
        4    0.000    0.000    0.000    0.000 /Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/random.py:217(randint)
        4    0.000    0.000    0.000    0.000 /Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/random.py:223(_randbelow)
        4    0.000    0.000    0.000    0.000 /Users/rizos/Desktop/crypto2/miller_rabin.py:6(randint)
        1    0.000    0.000    0.000    0.000 /Users/rizos/Desktop/crypto2/miller_rabin.py:12(find_s_t)
       11    0.000    0.000    0.000    0.000 /Users/rizos/Desktop/crypto2/miller_rabin.py:27(test_congruence)
        1    0.000    0.000    0.000    0.000 /Users/rizos/Desktop/crypto2/miller_rabin.py:44(<listcomp>)
        1    0.046    0.046 4077.982 4077.982 /Users/rizos/Desktop/crypto2/miller_rabin.py:41(test_miller_rabin)
        1    0.000    0.000 4077.982 4077.982 /Users/rizos/Desktop/crypto2/miller_rabin.py:83(<lambda>)
        4 4077.935 1019.484 4077.935 1019.484 /Users/rizos/Desktop/crypto2/utilities.py:34(pow_modulus)



Test Result:    *** PRIME ***
Approved 4-times

Process finished with exit code 0
```

Σχήμα 13: Τεστ πιστοποίησης πρώτου Fib_{104911}.

ΚΕΦΑΛΑΙΟ 12

Safe Primes

Να χρησιμοποιήσετε το τεστ των Miller-Rabin, για να παράγετε δύο πρώτους αριθμούς p, q όπου ο p να έχει 2048 bits και ο q να είναι της μορφής $2p + 1$. Ο χρόνος εκτέλεσης να είναι μικρότερος των 100 second.

12.1 Προσέγγιση Επίλυσης

Στη θεωρία αριθμών ένας πρώτος αριθμός p ονομάζεται Sophie Germain Prime εάν είναι εξίσου πρώτους και ο $2p+1$, ο οποίος ονομάζεται safe prime. Τους αριθμούς αυτούς τους χρησιμοποίησε η προαναφερθήσα στις μελέτες της για το τελευταίο θεώρημα του Fermat [4].

Το μέγεθος του αριθμού p που αναζητούμε είναι αρκετά μεγάλο 617 ψηφίων, ενώ ισάριθμο είναι και το εύρος αναζήτησης. Εάν αποκλέισουμε του άρτιους αριθμούς μειώνεται στο μισό το πλήθος των αριθμών που χρειάζεται να ελέγξουμε. Όμως αξιοσημείωτο είναι το υψηλό υπολογιστικό κόστος που απαιτεί ο αλγόριθμος των Miller-Rabin για την πιστοποιήση ισχυρών πιθανών πρώτων αριθμών. Συνεπώς για την ελαχιστοποίηση του χρόνου υπολογισμού απαιτείται η ελάχιστη δυνατή χρήση του, γι αυτό προσπαθήσαμε να εξαλήψουμε υποψήφιους πρώτους Germain πρωτού προβούμε στη χρήση του τεστ.

Η πρώτη τεχνική που θα μπορούσε να εφαρμοσθεί είναι *Το Κόσκινο του Ερατοσθένη*. Σύμφωνα με αυτήν, δοθέντος ενός ακέραιου αριθμού n, υπολογίζονται όλοι οι πρώτοι αριθμοί μέχρι τον n. Η διαδικασία εφαρμογής όμως έχει μεγάλες απαιτήσεις μνήμης, καθώς το πρώτο βήμα συνηστά τη δημιουργία ενός συνόλου Σ που περιέχει όλους τους αριθμούς από το 2 εώς το n οπότε προκύπτει $\Sigma = \{2, 3, 4, \ldots, n\}$. Κατόπιν αφαιρείται από το σύνολο το πρώτο 2 και τα πολλαπλάσια του, το 3 και τα πολλαπλασιά του συνεχίζοντας μέχρι το $\sqrt{n}$. Το τελευταίο προκύπτει καθώς έχει αποδειχθεί πως κάθε σύνθετος αριθμός $a > 1$ έχει έναν τουλάχιστον πρώτο διαιρέτη $d \leq \sqrt{a}$. Καθιστάται προφανές πως η εφαρμογή της διαδικασίας αυτής για την εύρεση των πρώτων αριθμών 2048 bits πέρα από το υπολογιστικό κόστος απαιτεί και πολύ μεγάλη δέσμευση μνήμης.

12.2 Επίλυση

Σύμφωνα με την υπόθεση του Riemann από την οποία προκύπτει η λύση στο πρόβλημα της κατανομής των πρώτων αριθμών, προκύπτει πως σ' ένα σύνολο ακεραίων Σ , η επιλογή ενός πρώτου αριθμού $p \epsilon \Sigma$ με $p \leq N$ όπου N το πλήθος των στοιχείων του Σ, απαιτεί 1024 προσπάθειες [5]. Αυτό συμβαίνει καθώς ένας πρώτος αριθμός p εμφανίζεται στο σύνολο Σ με πιθανότητα $1/\ln N$, εξαλείφοντας τους άρτιους αριθμούς προκύπτει $\ln 2^{2048}/2 = 1024$. Γι αυτό και η επιλογή των υποψήφιων πρώτων αριθμών p, Sophie Germain Prime, έγινε με εφαρμογή ενός ψευδοτυχαίου αλγορίθμου επιλογής ακέραιων αριθμών, που διαλέγει αριθμούς των 2048bits στο εύρος μεταξύ μικρότερου και μεγαλύτερου πρώτου των 2048bits , που έχουν προϋπολογίσθεί.

Στις δημοσιεύσεις [3] [1] των Wiener και Naccache αντίστοιχα παρουσιάσθηκαν κάποιοι περιορισμοί που πρέπει να πληρούν τα p και q. Ο Wiener αναφέρει πως $p \equiv 2 \mod 3$, και ο Naccache $p \equiv 3 \mod 4$, ενώ για το $q \equiv 2 \mod 3$ και $q \equiv 3 \mod 4$. Αυτές οι συνθήκες ελέγχονται πριν εκτελεσθεί οποιαδήποτε αναζήτηση του p, προς αποφυγή σπατάλης χρόνου.

Η μέθοδος κοσκινίσματος που προαναφέρθηκε εφαρμόζεται για τον υπολογισμό του συνόλου P_8 που ανήκουν όλοι οι πρώτοι αριθμοί μεγέθους εώς και 8bits, και του συνόλου P_{16} που απαρτίζεται από όλους τους πρώτους αριθμούς εως 16bits. Για κάθε $x \epsilon P_8$ ελέγχεται εάν $p \equiv (x-1)/2 \mod x$ [3] ή εαν ισχύει $\gcd((p-1)/2, x) = 1$ [1], όπου τα p εξαλείφονται και πάλι. Κατόπιν για το σύνολο P_{16} ελέγχεται εάν για κάποιο $y \epsilon P_{16}$ ισχύει πως $y|p$ ή το $y|q$, με $q = 2p + 1$.

Μετά την εφαρμογή των παραπάνω ακολουθεί το τεστ των Mieler-Rabin για το p. Εάν ο p είναι ισχυρός πιθανός πρώτος αριθμός γίνεται έλεγχος ώστε να βεβαιωθεί το ίδιο και για το q. Τέλος όταν καταλήξουμε στα p, q μπορούμε να επαληθεύσουμε εφαρμόζοντας τη μέθοδο πιστοποιήσης μερικές φορές ακόμη.

Στην εικόνα 7, που ακολουθεί παρουσιάζονται τα αποτελέσματα της αναζήτησης ενός safe prime που προκύπτει απο Germain πρώτο αριθμό των 2048bits, σε χρόνο 82 seconds. Όπως παρατηρείται έγινε έλεγχος 1401921 υποψήφιων p τα οποία απορρίφθηκαν, ενώ μόλις 13 από αυτά (όσο και το πλήθος των *) πέρασαν τους προελέγχους και εκτελέσθηκε το τεστ πιστοποίησης για τον q. Ο αλγόριθμος που αναπτύχθηκε είναι γραμμικός, ενώ με μία παράλληλη υλοποίηση θα είχαμε εκθετικά μικρότερο χρόνο αναζήτησης αφού παραλληλοποιείται πλήρως.

```
Run
Run:  11-Big_Primes_Gen
START
* * * * * * * * * * * * *
Number of tries: 1401921

         17761824 function calls in 81.985 seconds

   Random listing order was used

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
  1402906    0.158    0.000    0.158    0.000 {method 'bit_length' of 'int' objects}
      986    0.005    0.000    0.005    0.000 {built-in method builtins.hasattr}
       14    0.000    0.000    0.000    0.000 {built-in method builtins.print}
      985    0.002    0.000    0.002    0.000 {built-in method math.log2}
  1403318    1.295    0.000    1.295    0.000 {method 'getrandbits' of '_random.Random' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
      986    0.001    0.000    0.006    0.000 <frozen importlib._bootstrap>:997(_handle_fromlist)
  1402906    1.466    0.000    3.827    0.000 /Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/random.py:173(randrange)
  1402906    0.715    0.000    4.542    0.000 /Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/random.py:217(randint)
  1402906    0.908    0.000    2.361    0.000 /Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/random.py:223(_randbelow)
  4876141    5.514    0.000    5.514    0.000 /Users/rizos/Desktop/crypto2/11/11-Big_Primes_Gen.py:10(<lambda>)
    18571   19.792    0.001   19.792    0.001 /Users/rizos/Desktop/crypto2/11/11-Big_Primes_Gen.py:66(sievingP16)
   233623    5.251    0.000    7.571    0.000 /Users/rizos/Desktop/crypto2/11/11-Big_Primes_Gen.py:75(extraSievingP)
  1401921    0.792    0.000   11.557    0.000 /Users/rizos/Desktop/crypto2/11/11-Big_Primes_Gen.py:97(sievingP)
       13    0.000    0.000    0.000    0.000 /Users/rizos/Desktop/crypto2/11/11-Big_Primes_Gen.py:112(sievingQ)
  1401921    0.485    0.000    5.018    0.000 /Users/rizos/Desktop/crypto2/11/11-Big_Primes_Gen.py:120(<lambda>)
  1401921    0.546    0.000    0.546    0.000 /Users/rizos/Desktop/crypto2/11/11-Big_Primes_Gen.py:121(<lambda>)
  1401922    0.875    0.000    6.439    0.000 /Users/rizos/Desktop/crypto2/11/11-Big_Primes_Gen.py:116(random_field)
      985    0.004    0.000   43.599    0.044 /Users/rizos/Desktop/crypto2/11/11-Big_Primes_Gen.py:134(<lambda>)
        1    0.598    0.598   81.985   81.985 /Users/rizos/Desktop/crypto2/11/11-Big_Primes_Gen.py:133(pair_finder)
        1    0.000    0.000   81.985   81.985 /Users/rizos/Desktop/crypto2/11/11-Big_Primes_Gen.py:162(<lambda>)
      985   43.524    0.044   43.524    0.044 ../utilities.py:34(pow_modulus)
      985    0.017    0.000    0.025    0.000 ../miller_rabin.py:12(find_s_t)
      985    0.002    0.000    0.011    0.000 ../miller_rabin.py:6(randint)
     1965    0.002    0.000    0.002    0.000 ../miller_rabin.py:27(test_congruence)
      985    0.002    0.000    0.013    0.000 ../miller_rabin.py:44(<listcomp>)
      985    0.032    0.000   43.595    0.044 ../miller_rabin.py:41(miller_rabin)



P:
 17106653655212159311061317412691476659721945974287020726814283231092251317689044223937659395796191119216469696487506241870445336014764802
 4481035413017103725120903153883353360774515942766020504440247828488207488626467491661720279040338551982552918038199434050782669570130333
 2662169793601549337152907697897125519085033320781185723282578090801219161402789617318468507538314089953659863467660898073128921802868975
 3027157483244403720093906018583414279731256737387896116597146527800267416486661722446785595577036603202050879931360128631248863409623543
 1172617906635515060404267832001808397003400557795646782316358024894908939

Q:
 3421330731042431862212263482538295331944389194857404145362856646218450263537808844787531879159238238432939392975012483740890672029529604
 8962070826034207450241806307766706721549031885532041008880495656976414977252934983323440558080677103965105836076398868101565339140260666
 5324339587203098674305815395794251038170066641562371446565156181602438322805579234636937015076628179907319726935321796146257843605737950
 6054314966488807440187812037166828559462513474775792233194293055600534832973323444893571191154073206404101759862720257262497726819247086
 23452358132710301208085356640036167940068011155912935644632716049789817879

Process finished with exit code 0
```

Σχήμα 14: Search Germain Prime 2048bits, 82 sec.

ΚΕΦΑΛΑΙΟ 13

RSA

Δίνεται το δημόσιο κλειδί (N, e)=(11413, 19). Βρείτε το ιδιωτικό κλειδί και κατόπιν αποκρυπτογραφήστε το μήνυμα

$$C = (3203, 909, 3143, 5255, 5343, 3203, 909, 9958, 5278, 5343,$$
$$9958, 5278, 4674, 909, 9958, 792, 909, 4132, 3143, 9958, 3203, 5343, 792, 3143, 4443)$$

Υποθέστε ότι τα γράμματα στο αρχικό μήνυμα m, αναπαριστάνται από τις ASCII τιμές τους (δουλέψτε block by block το C).
Υποδ. Παραγοντοποιήστε N, κατόπιν υπολογίστε το $\varphi(N)$.

13.1 Επίλυση

Μετά από παραγοντοποίση του N προκύπτει ότι $N = 11413 = 101 \cdot 113$άρα έχουμε $p = 101$ και $q = 113$. Γνωρίζουμε πως $\varphi(n) = (p-1)(q-1) = (101-1)(113-1) = 11200$. Επίσης ισχύει πως

$$e \cdot d \equiv 1 \mod \varphi(n) \Rightarrow 19 \cdot d \equiv 1 \mod 11200$$

Επειδή ισχύει $\gcd(11200, 19) = 1$ υπάρχει μοναδική λύση. Σύμφωνα με το συντελεστές Bezout προκύπτει πως

- $11200 = 19 \cdot 589 + 9$
- $19 = 2 \cdot 9 + 1$

Οπότε προκύπτει $\gcd(11200, 19) = 1 = 19 - 2 \cdot 9 = 19 - 2 \cdot (11200 - 19 \cdot 589 = 19 - 2 \cdot 11200 + 19 \cdot 1178 = 19 \cdot 1179 + (-2) \cdot 11200$.
Αντίστροφος του $19 \mod 11200$ είναι η κλάση $1179 \mod 11200$.

13.2 Υλοποίηση

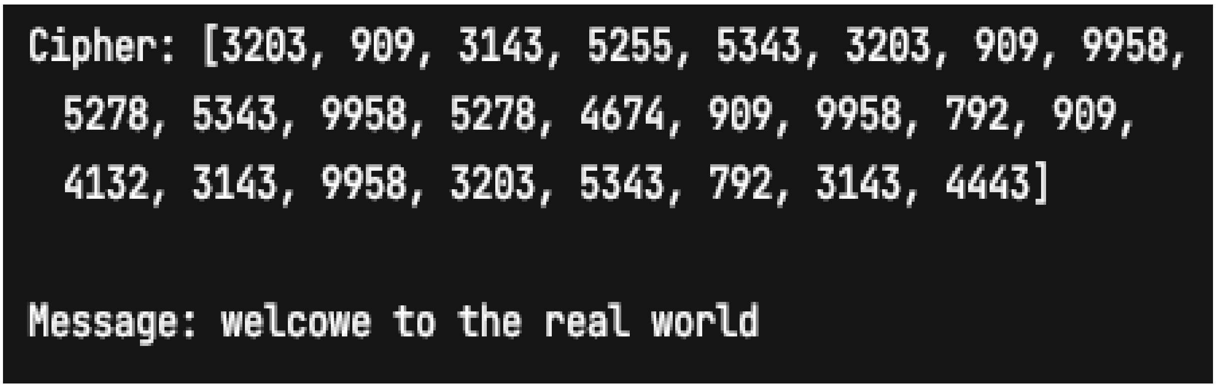
Αφού $d = e^{-1} \mod N \Rightarrow d = 1179$. Για το δημόσιο κλειδί pk ισχυεί πως $y = F(pk, x) \Rightarrow RSA_e(x) = x^e \mod N \Rightarrow RSA_{19}(x) = x^{19} \mod 11413$, όπου y το κρυπτογραφημένο μήνυμα και x το μήνυμα. Το μυστικό κλειδί γινεται αντιστοίχως $sk(d, N) = (1179, 11413)$. Από εδώ και στο εξής θα αναφερόμαστε με c στο κρυπτογραφημένο μήνυμα και m για το μήνυμα για λόγους κατανόησης. Ακολούθως λοιπόν συμβολίζεται με i ο αριθμός του block του μηνύματος όπου $i \in \mathbb{Z}$

$m_i = c_i^d \mod N \quad \Rightarrow^{d=1179} \quad m_i = c_i^{1179} \mod 11413 \quad \Rightarrow^{i=1} \quad m_1 = c_1^{1179} \mod 11413$
$\Rightarrow^{c_1=3202} \quad m_1 = 119$

13.3 Αποτέλεσμα

Προς αποκρυπτογράφηση του μηνύματος αναπτύχθηκε πρόγραμμα σε Python το οποίο υπολογίζει όλα τα m_i, όπου $i \in \mathbb{Z}_{i>0}$ και συμβολίζει το αντίστοιχο block μηνύματος. Κατόπιν αντικαθιστά κάθε αριθμό με την αντίστοιχη ASCII τιμή του. Τα αποτελέσματα παρουσιάζονται στην εικόνα 8.

welcome to the real world



```
Cipher: [3203, 909, 3143, 5255, 5343, 3203, 909, 9958,
  5278, 5343, 9958, 5278, 4674, 909, 9958, 792, 909,
  4132, 3143, 9958, 3203, 5343, 792, 3143, 4443]

Message: welcowe to the real world
```

Σχήμα 15: Decrypt RSA Ciphe.

ΚΕΦΑΛΑΙΟ 14

Επίθεση Wiener

Ας θεωρήσουμε $(N, e) = (194749497518847283, 50736902528669041)$ και το κρυπτογραφημένο κείμενο που δίνεται στο github.com/drazioti/ . . . , έχει προκύψει από το textbook-RSA (block-by-block) και κατόπιν κωδικοποιήθηκε. Εφαρμόστε την επίθεση του Wiener, για να βρείτε το κλεδί d. Υποθέτουμε ότι στο αρχικό κείμενο m κάθε χαρακτήρας έχει αντικατασταθεί από την ASCII τιμή του. Τέλος, βρείτε το αρχικό μήνυμα m.

14.1 Υλοποίηση

Μετά τη λήψη του δοθέντος αρχείου, αποκωδικοποείται το το κρυπτογραφημένο κείμενο, το οποίο είναι κωδικοποιημένο με τον αλγόριθμο base64 καθώς φέρει τα χαρακτηριστικά στο τέλος του μηνύματος. Το μηνυμα που προκύπτει εμφανίζεται στην εικόνα 9.

```
cipher: C=[47406263192693509,51065178201172223,30260565235128704,82385963334404268
8169156663927929,47406263192693509,17827597336696442,134434295894803806
112111571835512307,119391151761050882,30260565235128704,82385963334404268
134434295894803806,47406263192693509,45815320972560202,174632229312041248
30260565235128704,47406263192693509,119391151761050882,57208077766585306
134434295894803806,47406263192693509,119391151761050882,47406263192693509
112111571835512307,52882851026072507,119391151761050882,57208077766585306
119391151761050882,112111571835512307,8169156663927929,134434295894803806
57208077766585306,47406263192693509,185582105275050932,174632229312041248
134434295894803806,82385963334404268,172565386393443624,106356501893546401
8169156663927929,47406263192693509,10361059720610816,134434295894803806
119391151761050882,172565386393443624,47406263192693509,8169156663927929
52882851026072507,119391151761050882,8169156663927929,47406263192693509
45815320972560202,174632229312041248,30260565235128704,47406263192693509
52882851026072507,119391151761050882,111523408212481879,134434295894803806
47406263192693509,112111571835512307,52882851026072507,119391151761050882
57208077766585306,119391151761050882,112111571835512307,8169156663927929
134434295894803806,57208077766585306]
```

Σχήμα 16: Cipher textbook-RSA.

Το επόμενο βήμα είναι η παραγοντοποίηση του N, εφαρμόζοντας την επίθεση Wiener. Η επίθεση αυτή απαιτεί έναν αλγόριθμο υπολογισμού συνεχών κλασμάτων, ο οποίος και υλοποιήθηκε βάση του αλγορίθμου (12.1.1) του [5] ώστε να υπολογισθούν τα $\frac{e}{N}$. Οι ανάγκες του αλγορίμου απαιτούν όλα τα ανάγωγα κλάσματα $\frac{N_i}{D_i}$ έτσι αναπτύχθηκε κατάλληλος αλγόριθμος όπως ακόμη και ένας βοηθητικός αλγόριθμος που υλοποιεί πρόσθεση μεταξύ κλασμάτων. Στην εικόνα 10 παρουσιάζεται το συνεχές κλάσμα $\frac{e}{N}$, τα ανάγωγα $\frac{N_i}{D_i}$, τα φ_i, τα p, q που προέκυψαν μετά την εφαρμογή του Wiener και φυσικά το d και το κρυπτογραφημένο μήνυμα.

Just because you are a character doesn't mean that you have character

```
e-N: [0, 3, 1, 5, 5, 3, 2, 1, 1, 2, 1, 2, 1, 1, 1, 2]

Ni-Di: [(1, 3), (1, 4), (6, 23), (31, 119), (99, 380), (229, 879), (328, 1259), (557,
   2138), (1442, 5535), (1999, 7673), (5440, 20881), (7439, 28554), (12879, 49435),
  (20318, 77989), (53515, 205413)]

Phi: [152210707586007122, 202947610114676163, 194491459693231323, 194764238739084383,
   194747706675699349, 194749944640611733, 194749269157299764, 194749546869469317,
  194749483700543094, 194749501301889720, 194749496636238648, 194749497889987336,
  194749497360412612, 194749497554305041, 194749497507642599]
p: 441364039     q:441244597
d: 20881
message:  Just because you are a character doesn't mean that you have character
```

Σχήμα 17: Αποτέλεσμα της επίθεσης Wiener

Αναφορές

[1] David Naccache. Double-speed safe prime generation. *Gemplus Card International Applied Research Security Centre*, page 175, 2003.

[2] Sameer Punjal. Fibonacci algorithms. `https://eskeype.github.io/2018/04/20/7-Fibonacci-Algorithms`, April 2018.

[3] Michael J. Wiener. Safe prime generation with a combined sieve. *Cryptographic Clarity*, page 186, 2003.

[4] Wikipedia. Safe and Sophie Germain primes — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Safe%20and%20Sophie%20Germain%20primes&oldid=1065829680`, 2022. [Online; accessed 06-February-2022].

[5] Δραζιώτης Κωνσταντίνος. *Εισαγωγή στην Κρυπτογραφία*. Εκδόσεις Κάλλιπος, 2022.